**SQL**

**1.Types of commands and their examples.**

**Short answer:**

Types of commands are Data Definition Language (DDL), Data Manipulation Language (DML), Data Query Language (DQL), and Data Control Language (DCL)

**Long answer with examples:**

* Data Definition Language (DDL): DDL commands are used to define or modify the structure of database objects such as tables, indexes, and views.

Example: create table tablename;

* Data Manipulation Language (DML): DML commands are used to manipulate data within database objects like inserting, updating, or deleting records.

Example: INSERT INTO tablename (col1, col2, col3) VALUES (value1,value2,value3);

* Data Query Language (DQL): DQL commands are used to retrieve data from the database.

Example: SELECT \* FROM tablename WHERE col= value;

* Data Control Language (DCL): DCL commands are used to control access to data within the database.

Example: GRANT SELECT, INSERT ON employees TO 'user'@'localhost' IDENTIFIED BY 'password';

**2.What is Normalization and denormalization?**

**Short answer:**

Normalization aims to ensure data integrity and reduce redundancy in databases.

Denormalization aims to improve query performance by reintroducing redundancy and simplifying data retrieval

**Long answer:**

Normalization is the process of organizing data in a database to minimize redundancy and dependency by dividing large tables into smaller tables and defining relationships between them. The main goal of normalization is to eliminate data redundancy and anomalies such as insertion, update, and deletion anomalies. This process involves applying a series of rules called Normal Forms (e.g., First Normal Form (1NF), Second Normal Form (2NF), Third Normal Form (3NF), etc.). By normalizing a database, you ensure that each piece of data is stored exactly once, in exactly one place.

Denormalization, on the other hand, is the process of intentionally introducing redundancy into a normalized database to improve performance. This involves adding redundant copies of data or grouping data together into fewer tables to simplify queries and speed up data retrieval. Denormalization is typically done in situations where there are performance bottlenecks due to complex queries in highly normalized databases. It aims to optimize read performance at the expense of some additional storage space and increased complexity in data modification operations.

**3.Explain 1NF, 2NF, 3NF.**

**Short answer:**

1NF deals with eliminating repeating groups and ensuring atomicity, 2NF deals with ensuring all non-key attributes are fully functionally dependent on the primary key, and 3NF deals with eliminating transitive dependencies among non-key attributes. These normal forms help in organizing data to reduce redundancy and improve data integrity in a relational database.

**Long answer:**

First Normal Form (1NF):

* In 1NF, the data is organized into rows and columns where each cell contains a single value, meaning there are no repeating groups or arrays of values.
* Each attribute (or column) in a table must contain atomic (indivisible) values.
* There should be no repeating groups of columns.
* Example: Consider a table for storing student information. Each student can have multiple phone numbers, but in 1NF, these phone numbers must be stored in separate rows rather than in a single cell or column.

Second Normal Form (2NF):

* In addition to meeting 1NF requirements, a table must also ensure that all non-key attributes (attributes that are not part of the primary key) are fully functionally dependent on the entire primary key.
* This means that every non-key attribute must be dependent on the whole primary key, not just part of it.
* If any non-key attribute is functionally dependent on only a portion of the primary key, it violates 2NF.
* Example: In a table where student grades are stored with attributes (student\_id, course\_id, grade), if the grade is functionally dependent only on the course\_id and not on the student\_id, then it violates 2NF.

Third Normal Form (3NF):

* In addition to meeting 2NF requirements, a table must also ensure that all non-key attributes are not transitively dependent on the primary key.
* This means that if a non-key attribute is dependent on another non-key attribute, it should be moved to a separate table.
* Transitive dependency occurs when a non-key attribute is dependent on another non-key attribute which, in turn, is dependent on the primary key.
* Example: Consider a table where employee details are stored with attributes (employee\_id, department\_id, department\_name). Here, department\_name is transitively dependent on the employee\_id through department\_id. To adhere to 3NF, department\_name should be moved to a separate table with department\_id as its primary key.

**4.Share use case where you had to do denormalization in database.**

One common use case where denormalization is employed is in optimizing performance for read-heavy applications, especially when dealing with complex queries that involve multiple joins across normalized tables. Example:

Use Case: E-commerce Product Catalog

* Consider an e-commerce platform with a highly normalized database schema to manage its product catalog. The database schema includes tables such as products, categories, brands, and inventory.
* The products table stores basic information about each product, such as its name, description, price, and unique identifier (product\_id).
* The categories table contains different product categories with their respective identifiers (category\_id).
* The brands table stores information about different brands with their identifiers (brand\_id).
* The inventory table tracks the available quantity of each product in stock.
* In this normalized schema, to retrieve detailed product information along with its category and brand information, one would typically need to perform multiple JOIN operations across these tables. While normalization ensures data integrity and reduces redundancy, it can lead to increased query complexity and longer execution times, especially for queries involving multiple joins.
* To improve the performance of queries fetching product information, especially for product listings or searches, denormalization can be applied. This could involve creating a separate denormalized table, say product\_details, which consolidates data from the normalized tables.
* The product\_details table might include columns such as product\_id, product\_name, description, price, category\_name, brand\_name, and quantity\_in\_stock. The category\_name and brand\_name columns would store denormalized data retrieved from the categories and brands tables, respectively.

By denormalizing the data in this manner, queries fetching product details no longer require multiple joins across normalized tables. Instead, they can directly retrieve all necessary information from the denormalized product\_details table, resulting in simpler and faster queries, especially for read-heavy operations such as product listings, searches, and recommendations.

However, it is important to note that denormalization comes with trade-offs, such as increased storage space and complexity in data modification operations (e.g., updates, inserts, and deletes). Therefore, it should be carefully considered based on the specific performance requirements and trade-offs of the application.

**5.What is primary key and foreign key?**

**Short answer:**

A primary key uniquely identifies each record in a table.

A foreign key establishes relationships between tables by referencing the primary key or a unique key in another table. These constraints help maintain data integrity and enforce the integrity of relationships between tables in a relational database

**Long answer:**

**Primary Key:**

A primary key is a column or a set of columns in a database table that uniquely identifies each row or record in that table. The primary key constraint ensures that the values in the specified column(s) are unique and not null. In other words, it uniquely identifies each record within the table and serves as a reference point for other tables to establish relationships. Typically, tables have one primary key, although composite primary keys, which consist of multiple columns, are also possible.

Example:

In a table storing information about employees, the employee ID column could be designated as the primary key. Each employee would have a unique ID, and this column would ensure that no two employees have the same ID.

**Foreign Key:**

A foreign key is a column or a set of columns in a table that establishes a link between data in two tables. It creates a relationship between two tables by referencing the primary key or a unique key in another table. The foreign key constraint ensures that the values in the foreign key column(s) must match the values in the referenced primary key column(s) or unique key column(s) in the related table, or be null if the relationship is optional.

Example:

In a database schema containing two tables, one for employees and another for departments, the department ID column in the employees table could serve as a foreign key. It would reference the primary key (department ID) in the departments table. This foreign key constraint ensures that each employee is associated with a valid department by referencing an existing department ID in the departments table.

**6.what is alternate and candidate key?**

**Short answer:**

Candidate keys are columns or sets of columns that uniquely identify records in a table. Alternate keys are candidate keys that are not selected as the primary key. Both candidate keys and alternate keys play important roles in maintaining data integrity within a relational database.

**Long answer:**

**Candidate Key:**

A candidate key is a column or a set of columns in a relational database table that uniquely identifies each record or row within that table. Like a primary key, a candidate key must satisfy two conditions:

Uniqueness: Each value or combination of values in the candidate key must be unique across all rows in the table.

Irreducibility: No subset of the candidate key should have the uniqueness property. In other words, removing any attribute from the candidate key would result in a loss of uniqueness.

Tables may have multiple candidate keys, but only one of them will be chosen as the primary key. The remaining candidate keys are called alternate keys.

**Alternate Key:**

An alternate key, also known as a secondary key, is any candidate key that is not selected as the primary key. While a primary key uniquely identifies each record in a table, alternate keys provide additional unique identifiers for the records. Although alternate keys are not chosen as the primary key, they still maintain uniqueness within the table.

Example:

Consider a table named "Employees" with attributes such as "Employee\_ID," "Email," and "Social\_Security\_Number." Both "Employee\_ID" and "Social\_Security\_Number" could individually serve as candidate keys for the table because they uniquely identify each employee. However, the primary key might be chosen as "Employee\_ID" due to its simplicity and stability.

In this example, "Social\_Security\_Number" becomes an alternate key. It retains the uniqueness constraint and can still be used to uniquely identify employees, even though it's not designated as the primary key.

**7.What are window functions?**

**Short Answer:**

Window functions, also known as windowing or analytic functions, are a category of SQL functions that operate on a subset of rows related to the current row within a query result set. These functions allow you to perform calculations across a "window" of rows defined by a set of rows or range of rows relative to the current row.

**Long answer:**

Key features of window functions include:

Partitioning: Window functions can be partitioned into groups of rows based on one or more columns. Each partition acts as a separate window for the function calculation.

Ordering: Within each partition, rows can be ordered based on one or more columns to define the order in which the window function operates.

Frame Specification: Window functions can operate on a specific range of rows within each partition, known as the "window frame." This frame can be defined by a fixed number of rows preceding or following the current row, or by a range of rows between specified bounds.

Commonly used window functions include:

* ROW\_NUMBER(): Assigns a unique sequential integer to each row within a partition, based on the specified ordering.
* RANK(): Assigns a unique rank to each row within a partition, with gaps in the ranking for duplicate values.
* DENSE\_RANK(): Similar to RANK(), but without gaps in the ranking for duplicate values.
* LEAD(): Returns the value of a specified column from the next row within the partition.
* LAG(): Returns the value of a specified column from the previous row within the partition.
* SUM(), AVG(), MIN(), MAX(): Aggregates values within the window frame.

Window functions are particularly useful for performing complex analytical tasks, such as calculating moving averages, cumulative sums, rank-based aggregations, and identifying top-N rows within groups. They provide a powerful and flexible mechanism for data analysis within SQL queries, enabling efficient computation of results without the need for subqueries or self-joins.

**8.Explain Ranking Functions? Given a small table, write the output.**

Ranking functions are a category of window functions in SQL that assign a rank to each row in a result set based on a specified ordering. These functions are used to determine the relative position of rows within a partition or the entire result set.

Commonly used ranking functions include:

ROW\_NUMBER(): Assigns a unique sequential integer to each row, starting from 1, within a partition.

RANK(): Assigns a unique rank to each distinct row, with the same rank for rows with equal values. Gaps may occur in the ranking if there are ties.

DENSE\_RANK(): Similar to RANK(), but without gaps in the ranking for rows with equal values.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ID | Name | Score | Row Number | Rank | Dense Rank |
| 1 | Alice | 99 | 1 | 1 | 1 |
| 2 | Bob | 97 | 2 | 2 | 2 |
| 3 | Charlie | 97 | 3 | 2 | 2 |
| 4 | David | 92 | 4 | 4 | 3 |
| 5 | Emily | 91 | 5 | 5 | 4 |

**9.Types of Joins? With example and usecase. All the number of records return and exact records.**

**Short Answer:**

Types of joins in SQL include INNER JOIN, LEFT JOIN, RIGHT JOIN, and FULL JOIN. Each join type serves a specific purpose for combining data from multiple tables based on common keys.

**Long answer:**

To provide the output for all joins (INNER JOIN, LEFT JOIN, RIGHT JOIN, and FULL JOIN) between Table A and Table B based on their respective IDs, let's first define the tables:

Table A:

|  |  |
| --- | --- |
| id | name |
| 1 | apple |
| 1 | apple |
| 1 | apple |
| 1 | apple |
| 2 | Banana |
| 2 | Banana |
| 3 | Guava |

Table B:

|  |  |
| --- | --- |
| id | color |
| 1 | Red |
| 2 | Yellow |
| 3 | Green |

Now, let us provide the output for each type of join:

1. INNER JOIN:

- Inner join returns only the records where there is a match between the IDs in both tables.

- Number of records returned: 3 (one for each matching ID: 1, 2, and 3).

- Code:

SELECT A.id, A.name, B.color FROM TableA A INNER JOIN TableB B ON A.id = B.id;

2. LEFT JOIN:

- Left join returns all records from Table A and the matched records from Table B.

- Number of records returned: 7 (all records from Table A).

- Code:

SELECT A.id, A.name, B.color FROM TableA A LEFT JOIN TableB B ON A.id = B.id;

3. RIGHT JOIN:

- Right join returns all records from Table B and the matched records from Table A.

- Number of records returned: 3 (all records from Table B).

- Code:

SELECT A.id, A.name, B.color FROM TableA A RIGHT JOIN TableB B ON A.id = B.id;

4. FULL JOIN:

- Full join returns all records from both Table A and Table B.

- Number of records returned: 7 (the combined total of records from both tables).

- Code:

SELECT A.id, A.name, B.color FROM TableA A FULL JOIN TableB B ON A.id = B.id;

These outputs illustrate the differences in results obtained from each type of join operation between Table A and Table B based on their IDs.

**10.Use case when self join is required.**

Self joins are essential in scenarios where you need to establish relationships between rows within the same table, such as hierarchical data structures like organizational charts or employee-manager relationships.

Use case:

A common use case for a self join is when you have hierarchical data stored in a single table and you need to query relationships between rows within that table. Here's an example:

Consider a table named "Employees" with the following columns: EmployeeID, Name, ManagerID.

EmployeeID is a unique identifier for each employee.

Name stores the name of each employee.

ManagerID represents the EmployeeID of the employee's manager.

Now, suppose you want to retrieve a list of employees along with the names of their managers. In this scenario, you would need to perform a self join on the "Employees" table to establish the relationship between employees and their managers.

**11.What is subquery?**

**Short Answer:**

A subquery, also known as an inner query or nested query, is a query nested within another SQL query. It allows you to perform a query within another query to retrieve data or perform additional operations based on the results of the outer query.

**Long Answer:**

Subqueries can appear in various parts of a SQL statement, including the SELECT, FROM, WHERE, HAVING, and INSERT statements. The result of a subquery can be used in comparison operations, filtering conditions, or as a source of data for other operations within the outer query.

Example:

Let's say we have two tables, "Orders" and "Customers," and we want to retrieve the names of customers who have placed orders. We can use a subquery within the WHERE clause to achieve this:

**SELECT Name FROM Customers WHERE CustomerID IN (SELECT CustomerID FROM Orders);**

In this example:

* The inner query retrieves CustomerIDs from the "Orders" table, representing customers who have placed orders.
* The outer query selects names from the "Customers" table where the CustomerID matches those retrieved by the inner query.
* The IN keyword is used to check if the CustomerID from the outer query exists in the set of CustomerIDs retrieved by the subquery.

Subqueries provide a powerful way to filter, manipulate, or correlate data within SQL statements, allowing for more complex and flexible queries. They are commonly used for tasks such as filtering data based on conditions, performing calculations, or generating dynamic result sets.

**12.What is corelated subquery?**

**Short Answer:**

A correlated subquery is a type of subquery in SQL where the inner query references one or more columns from the outer query.

**Long Answer:**

Unlike a non-correlated subquery, which can be evaluated independently of the outer query, a correlated subquery is dependent on the outer query for its execution.

In a correlated subquery, the inner query is executed repeatedly for each row processed by the outer query. The result of the subquery depends on the current row being processed in the outer query, as it can reference values from that row.

Example:

Suppose we have two tables, "Orders" and "Customers," where "Orders" contains order information including CustomerID, and "Customers" contains customer information including CustomerID. We want to retrieve the names of customers who have placed orders.

Here is an example of a correlated subquery to achieve this:

SELECT Name FROM Customers c WHERE EXISTS (SELECT 1 FROM Orders o WHERE o.CustomerID = c.CustomerID);

In this example:

* The outer query selects the Name column from the "Customers" table, aliased as 'c'.
* The inner query checks if there exists at least one row in the "Orders" table, aliased as 'o', where the CustomerID matches the CustomerID from the current row being processed in the outer query.
* The correlation occurs through the reference to c.CustomerID, where 'c' represents the alias of the "Customers" table from the outer query.

The correlated subquery is executed once for each row in the "Customers" table, and for each row, it checks if there is a corresponding record in the "Orders" table. If such a record exists, the customer's name is included in the result set.

Correlated subqueries are useful for situations where the filtering or condition depends on values from the outer query, allowing for more complex data retrieval and analysis. However, they can impact performance, especially if the inner query involves large datasets or complex operations. Therefore, they should be used judiciously and optimized carefully.

**13.What is CTE?**

**Short Answer:**

A Common Table Expression (CTE) is a temporary named result set that you can reference within a SELECT, INSERT, UPDATE, or DELETE statement in SQL. CTEs are defined using the WITH clause and offer a way to create named, temporary result sets that exist only for the duration of the query.

**Long Answer:**

CTEs are particularly useful for:

* Improving Readability: They allow you to break down complex queries into smaller, more manageable parts, improving the readability and maintainability of your SQL code.
* Avoiding Repetition: CTEs allow you to define a query once and reference it multiple times within the same SQL statement, eliminating the need to repeat complex subqueries.
* Recursive Queries: CTEs can be recursive, enabling you to perform hierarchical or recursive queries easily.

WITH cte\_name (column1, column2, ...) AS ( SELECT column1, column2, ...FROM your\_table WHERE condition)

SELECT \* FROM cte\_name;

In this example:

We define a CTE named "OrderCounts" that calculates the total number of orders for each customer by grouping the orders by CustomerID and counting them.

We then use this CTE in the main query to join the "Customers" table with the "OrderCounts" CTE to retrieve customer information along with their total order counts.

CTEs provide a powerful and flexible way to work with temporary result sets in SQL, enhancing the readability and maintainability of your queries.

**14.What is derived table?**

**Short Answer:**

A derived table, also known as an inline view or subquery factoring, is a temporary table that is derived from the result of a SELECT statement. It is commonly used within the FROM clause of a SQL query and allows you to treat the result of a subquery as if it were a physical table in the database.

**Long Answer:**

Derived tables are useful when you need to perform complex transformations, calculations, or filtering operations on data before using it in the main query. They can simplify query logic and improve readability by breaking down complex queries into smaller, more manageable parts.

Here's the basic syntax for using a derived table:

SELECT \* FROM (

-- Subquery defining the derived table

SELECT column1, column2, ...

FROM your\_table

WHERE condition

) AS derived\_table\_alias;

In this syntax:

- The subquery enclosed within parentheses defines the derived table.

- You can specify column names and apply filtering conditions within the subquery as needed.

- The AS keyword is used to assign an alias to the derived table, allowing you to refer to it in the outer query.

Example:

Suppose we have a table named "Orders" containing order information such as OrderID, CustomerID, and OrderDate. We want to retrieve a list of customers along with the total number of orders they have placed. We can use a derived table to calculate the total number of orders for each customer before joining it with the "Customers" table.

SELECT c.CustomerID, c.CustomerName, o.TotalOrders

FROM Customers c

JOIN (

SELECT CustomerID, COUNT(\*) AS TotalOrders

FROM Orders

GROUP BY CustomerID

) AS o ON c.CustomerID = o.CustomerID;

In this example:

- The inner subquery calculates the total number of orders for each customer by grouping the orders by CustomerID and counting them.

- The result of the subquery is treated as a derived table with the alias "o".

- The outer query joins the "Customers" table with the derived table based on the common CustomerID column, allowing us to retrieve customer information along with their total order counts.

Derived tables provide a flexible and powerful way to manipulate data within SQL queries, allowing you to perform complex operations and transformations on data before using it in the main query.

**15.Find third highest employee based on salary?**

To find the third highest employee based on salary, you can use the following SQL query:

SELECT EmployeeID, EmployeeName, Salary

FROM Employees

ORDER BY Salary DESC

LIMIT 1 OFFSET 2;

In this query:

- We select the EmployeeID, EmployeeName, and Salary columns from the Employees table.

- We use the ORDER BY clause to sort the results by the Salary column in descending order, so that the highest salary comes first.

- We use the LIMIT 1 OFFSET 2 clause to skip the first two rows and retrieve the third row, which corresponds to the third highest salary.

This query will return the details of the employee with the third highest salary in the Employees table. Adjust the table and column names according to your database schema.

**16.Find third highest employee based on salary per department?**

To find the third highest employee based on salary per department, you can use a combination of the SQL ORDER BY clause, the PARTITION BY clause for window functions (such as ROW\_NUMBER()), and a subquery. Here's how you can do it:

```sql

WITH RankedEmployees AS (

SELECT EmployeeID, EmployeeName, Salary, DepartmentID,

ROW\_NUMBER() OVER (PARTITION BY DepartmentID ORDER BY Salary DESC) AS SalaryRank

FROM Employees

)

SELECT EmployeeID, EmployeeName, Salary, DepartmentID

FROM RankedEmployees

WHERE SalaryRank = 3;

```

In this query:

- We first use a Common Table Expression (CTE) named RankedEmployees to assign a rank to each employee within their respective department based on their salary. The ROW\_NUMBER() function is used for this purpose, with the PARTITION BY clause partitioning the ranking by the DepartmentID and the ORDER BY clause ordering the employees by salary in descending order within each department.

- Then, in the main query, we select the EmployeeID, EmployeeName, Salary, and DepartmentID from the RankedEmployees CTE, filtering only the rows where the SalaryRank equals 3. This will give us the third highest employee based on salary per department.

Adjust the column and table names as per your schema. This query will provide you with the details of the third highest salary earner in each department.

**17.How to find duplicate values in a single column?**

To find duplicate values in a single column, you can use a GROUP BY clause along with the HAVING clause to filter out only those values that appear more than once. Here's an example SQL query:

```sql

SELECT column\_name, COUNT(\*)

FROM table\_name

GROUP BY column\_name

HAVING COUNT(\*) > 1;

```

In this query:

- Replace `column\_name` with the name of the column in which you want to find duplicates.

- Replace `table\_name` with the name of the table containing the column.

- We use the GROUP BY clause to group the rows based on the values in the specified column.

- We use the COUNT(\*) function to count the number of occurrences of each value in the column.

- We use the HAVING clause to filter out only those values that have a count greater than 1, indicating that they are duplicates.

This query will return all the values in the specified column that appear more than once, along with the count of their occurrences. Adjust the column and table names according to your database schema.

**18.How to find duplicate values in a multiple column?**

To find duplicate values across multiple columns, you can use the GROUP BY clause along with the HAVING clause to filter out only those rows that have duplicate values across those columns. Here's an example SQL query:

```sql

SELECT column1, column2, ..., COUNT(\*)

FROM table\_name

GROUP BY column1, column2, ...

HAVING COUNT(\*) > 1;

```

In this query:

- Replace `column1`, `column2`, etc., with the names of the columns across which you want to find duplicates.

- Replace `table\_name` with the name of the table containing the columns.

- We use the GROUP BY clause to group the rows based on the values in the specified columns.

- We use the COUNT(\*) function to count the number of occurrences of each combination of values in the specified columns.

- We use the HAVING clause to filter out only those combinations of values that have a count greater than 1, indicating that they are duplicates.

This query will return all the rows in the table where the combination of values across the specified columns appears more than once, along with the count of their occurrences. Adjust the column and table names according to your database schema.

**19.What are ACID properties? give example for each property**

**Short answer:**

ACID properties ensure that database transactions are Atomic (indivisible), Consistent (maintaining data integrity), Isolated (executed independently), and Durable (persistently stored), guaranteeing reliability and integrity of data operations.

**Long answer:**

ACID properties are a set of four key characteristics that guarantee the reliability and consistency of database transactions. The acronym stands for:

1. Atomicity: This property ensures that a transaction is treated as a single unit of work, meaning that either all of its operations are successfully completed or none of them are. If any part of the transaction fails, the entire transaction is rolled back to its initial state.

Example: Consider a banking transaction where funds are transferred from one account to another. Atomicity ensures that if the withdrawal from one account succeeds but the deposit into the other account fails (due to network issues or other reasons), the entire transaction is rolled back, ensuring that neither account is left in an inconsistent state.

2. Consistency: Consistency ensures that the database remains in a valid state before and after the transaction. All constraints, such as foreign key constraints, integrity constraints, and business rules, must be maintained during the execution of a transaction.

Example: In a database containing student records, if a transaction involves enrolling a student in a course, the database must ensure that the student exists in the student table before allowing enrollment in the course. Otherwise, the enrollment transaction should fail to maintain consistency.

3. Isolation: Isolation ensures that the concurrent execution of transactions does not interfere with each other, i.e., each transaction is executed as if it were the only transaction in the system. Isolation prevents transactions from accessing each other's intermediate states and ensures that the result of a transaction is consistent with the state of the database at the time the transaction started.

Example: Consider two transactions simultaneously updating the same bank account balance. Isolation ensures that each transaction sees the account balance as it was before any other transaction started, preventing conflicts and maintaining data integrity.

4. Durability: Durability guarantees that once a transaction is committed, its changes are permanent and will survive system failures such as power outages, crashes, or hardware failures. The changes made by a committed transaction are stored persistently and cannot be undone.

Example: After a successful funds transfer transaction, the changes to account balances must be durable, meaning that even if the database server crashes immediately after the transaction commits, the transferred funds remain intact and are not lost.

**20.Diff between union and union all**

The main difference between UNION and UNION ALL in SQL lies in how they handle duplicate rows:

1. UNION:

- The UNION operator is used to combine the result sets of two or more SELECT statements.

- It removes duplicate rows from the combined result set.

- Only distinct rows are returned in the final result set.

Example:

```

SELECT column1 FROM table1

UNION

SELECT column1 FROM table2;

```

2. UNION ALL:

- The UNION ALL operator also combines the result sets of two or more SELECT statements.

- Unlike UNION, it does not remove duplicate rows from the combined result set.

- All rows from all SELECT statements are included in the final result set, including duplicate rows.

Example:

```

SELECT column1 FROM table1

UNION ALL

SELECT column1 FROM table2;

```

In summary:

- UNION removes duplicate rows from the combined result set, whereas UNION ALL includes all rows, including duplicates.

- UNION is generally slower than UNION ALL because it requires the database engine to perform an additional step to remove duplicates.

- If you are sure that there are no duplicate rows or if you want to include duplicate rows in the result set, you can use UNION ALL for better performance. If you want to remove duplicates, use UNION.

**21.Diff between primary key and unique key**

**Short Answer:**

The primary key uniquely identifies each record in a table and does not allow null values, whereas the unique key ensures uniqueness but allows for one null value and does not necessarily serve as the main identifier for the table. Additionally, a table can have only one primary key but multiple unique keys.

**Long Answer:**

Both primary key and unique key are constraints used in database tables to enforce data integrity, but they have some key differences:

1. Purpose:

- Primary Key: The primary key uniquely identifies each record in a table. It serves as the main identifier for the table and ensures that each row has a unique identity.

- Unique Key: The unique key constraint also enforces uniqueness, but it allows for only one null value. It can be used to ensure that a column (or a combination of columns) contains only unique values, but it does not necessarily serve as the primary identifier for the table.

2. Number of Columns:

- Primary Key: A table can have only one primary key. It can be composed of one or multiple columns.

- Unique Key: A table can have multiple unique keys. Each unique key can be composed of one or multiple columns.

3. Null Values:

- Primary Key: Primary key columns cannot contain null values. Each primary key value must be unique and not null.

- Unique Key: Unique key columns can contain null values, except for one null value. If a unique key is composed of multiple columns, the combination of values must be unique, but individual columns can contain null values.

4. Automatic Indexing:

- Primary Key: In many database systems, defining a primary key automatically creates a unique index on the primary key column(s) to enforce uniqueness and optimize querying.

- Unique Key: Defining a unique key also typically creates a unique index to enforce uniqueness but does not have the same significance as a primary key index.

**22.Diff between truncate and delete**

**Short Answer:**

TRUNCATE is a faster operation that removes all rows from a table but cannot be rolled back.

DELETE is a slower operation that selectively removes rows based on conditions and can be rolled back within a transaction.

The choice between TRUNCATE and DELETE depends on the specific requirements of the task at hand and the desired outcome.

**Long Answer:**

Both TRUNCATE and DELETE are SQL commands used to remove data from a table, but they differ in functionality, speed, and the level of control they provide:

1. Functionality:

- TRUNCATE: TRUNCATE is a DDL (Data Definition Language) command used to remove all rows from a table. It is a faster operation compared to DELETE because it deallocates data pages and does not generate any rollback information. TRUNCATE resets identity columns to their seed value and resets the identity counter.

- DELETE: DELETE is a DML (Data Manipulation Language) command used to selectively remove rows from a table based on specified conditions. It is slower than TRUNCATE because it logs individual row deletions, generates rollback information, and maintains integrity constraints. DELETE does not reset identity columns.

2. Logging and Rollback:

- TRUNCATE: TRUNCATE is minimally logged, meaning it does not log individual row deletions. As a result, it cannot be rolled back, and the deleted data cannot be recovered.

- DELETE: DELETE is fully logged, meaning each deleted row is logged in the transaction log. This allows for rollback of individual deletions and recovery of deleted data if needed.

3. Speed:

- TRUNCATE: TRUNCATE is generally faster than DELETE, especially for large tables, because it does not generate log entries for each row deletion and does not activate triggers.

- DELETE: DELETE is slower compared to TRUNCATE because it logs each row deletion and activates triggers associated with the DELETE operation.

4. Transaction Control:

- TRUNCATE: TRUNCATE cannot be rolled back within a transaction. Once executed, the data is permanently removed from the table.

- DELETE: DELETE can be rolled back within a transaction. Changes made by DELETE can be undone using a ROLLBACK statement before the transaction is committed.

**23.Diff between having and where**

**Short Answer:**

- WHERE is used to filter individual rows based on conditions before any aggregation is performed.

- HAVING is used to filter groups of rows based on aggregate values after the aggregation is performed, typically with GROUP BY.

- WHERE is used with SELECT, UPDATE, and DELETE statements, while HAVING is used only with SELECT statements that include GROUP BY.

**Long Answer:**

The main difference between the HAVING and WHERE clauses in SQL lies in their usage and the stage at which they filter data:

1. WHERE Clause:

- The WHERE clause is used to filter rows before the aggregation is performed in a query.

- It is applied to individual rows and filters the rows that are included in the result set based on specified conditions.

- WHERE clause filters rows based on column values, typically before any aggregation functions are applied.

- It is used with SELECT, UPDATE, and DELETE statements.

Example:

```sql

SELECT column1, column2

FROM table\_name

WHERE condition;

```

2. HAVING Clause:

- The HAVING clause is used to filter rows after the aggregation is performed in a query, specifically with GROUP BY.

- It is applied to grouped rows and filters the groups (or aggregated values) that are included in the result set based on specified conditions.

- HAVING clause filters groups based on aggregate values, such as COUNT(), SUM(), AVG(), etc.

- It is used with SELECT statements that include GROUP BY clauses.

Example:

```sql

SELECT column1, COUNT(\*)

FROM table\_name

GROUP BY column1

HAVING COUNT(\*) > 1;

```

**24.SQL query execution order.**

The execution order of SQL queries typically follows a specific sequence of steps:

1. FROM: The FROM clause specifies the tables from which data will be retrieved. If there are multiple tables involved, the database system performs any necessary joins to combine the data.

2. WHERE: The WHERE clause filters the rows of data based on specified conditions. Only rows that satisfy the conditions specified in the WHERE clause are included in the result set.

3. GROUP BY: If a GROUP BY clause is present, the data is then grouped into sets based on the specified grouping columns. Aggregate functions (such as COUNT(), SUM(), AVG(), etc.) can be applied to each group.

4. HAVING: The HAVING clause filters the grouped data based on specified conditions. Only groups that satisfy the conditions specified in the HAVING clause are included in the result set. This step is performed after the GROUP BY operation.

5. SELECT: The SELECT clause specifies which columns or expressions will be included in the result set. It transforms the rows of filtered and grouped data into the final result set.

6. ORDER BY: If an ORDER BY clause is present, the result set is sorted based on the specified columns or expressions. This step is performed after the SELECT operation.

7. LIMIT/OFFSET: If a LIMIT or OFFSET clause is present, it limits the number of rows returned by the query and specifies the starting point for the result set.

It is important to note that not all SQL queries include all of these clauses, and the specific order of execution may vary depending on the structure of the query. Additionally, some database systems may optimize query execution by reordering or combining operations to improve performance.

**25.What are indexes? Types of Indexes and their differences.**

**Short Answer:**

Indexes in databases are data structures that improve the speed of data retrieval operations on tables by providing quick access to specific rows. They serve as a roadmap to data stored in tables, allowing database systems to locate rows more efficiently. Indexes are created on one or more columns of a table and can significantly enhance the performance of SELECT, UPDATE, DELETE, and JOIN operations.

**Long Answer:**

Types of Indexes:

1. Single-Column Index: This type of index is created on a single column of a table. It speeds up searches and comparisons based on that column.

2. Composite Index (or Multi-Column Index): A composite index is created on multiple columns of a table. It allows for faster retrieval of rows based on the combination of values in the indexed columns.

3. Unique Index: A unique index ensures that the values in the indexed column(s) are unique across the table. It enforces uniqueness and can improve the performance of queries that check for duplicates.

4. Primary Key: A primary key is a special type of unique index that uniquely identifies each row in a table. It ensures data integrity and provides a fast way to retrieve individual rows.

5. Clustered Index: A clustered index determines the physical order of data rows in a table based on the indexed column(s). The leaf nodes of a clustered index contain the actual data pages, and the rows are stored in the order specified by the index. Each table can have only one clustered index.

6. Non-Clustered Index: A non-clustered index stores a separate copy of the indexed column(s) along with a reference to the corresponding data rows. It does not affect the physical order of data rows in the table and is typically used for improving the performance of searches and queries that do not require accessing all columns.

Differences between Indexes:

1. Functionality: Each type of index serves a specific purpose in optimizing data retrieval operations. Single-column and composite indexes speed up searches based on column values, while unique indexes enforce uniqueness. Primary keys uniquely identify rows, and clustered indexes determine the physical order of rows.

2. Uniqueness: Unique indexes ensure that the values in the indexed column(s) are unique, whereas primary keys enforce both uniqueness and non-nullability of indexed columns.

3. Physical Order: Clustered indexes determine the physical order of data rows in a table based on the indexed column(s), whereas non-clustered indexes do not affect the physical order and store a separate copy of the indexed column(s).

4. Storage Overhead: Clustered indexes require additional storage space to maintain the physical order of rows, while non-clustered indexes store a separate copy of indexed columns along with references to data rows.

5. Performance Impact: The choice of index type can impact the performance of data retrieval operations. Clustered indexes are beneficial for range scans and point lookups, while non-clustered indexes are suitable for covering queries and searching based on non-key columns.

**26.What is surrogate key? Give example where you used it and how.**

**Short Answer:**

A surrogate key is a unique identifier assigned to each row in a database table to serve as the primary key, typically in cases where the natural key (a key composed of one or more existing columns in the table) is either absent, unreliable, or not suitable for use as a primary key. Surrogate keys are often implemented as auto-incrementing integers or GUIDs (Globally Unique Identifiers).

Example:

Let's consider a scenario where we have a table to store information about students in a school. The natural key for this table might be the combination of the student's first name, last name, and date of birth, as these attributes together uniquely identify each student. However, using this composite natural key as the primary key might not be efficient due to its length and complexity.

In this case, we can introduce a surrogate key, such as an auto-incrementing integer, to serve as the primary key for the table. This surrogate key would be a simple, unique identifier for each student record, making it easier and more efficient to reference and query student data.

Here's how we might create and use a surrogate key in SQL:

```sql

CREATE TABLE Students (

StudentID INT PRIMARY KEY AUTO\_INCREMENT,

FirstName VARCHAR(50),

LastName VARCHAR(50),

DateOfBirth DATE,

-- Other columns

);

INSERT INTO Students (FirstName, LastName, DateOfBirth)

VALUES ('John', 'Doe', '2000-01-01');

INSERT INTO Students (FirstName, LastName, DateOfBirth)

VALUES ('Jane', 'Smith', '2001-05-10');

```

In this example:

- The `StudentID` column is a surrogate key, defined as an auto-incrementing integer primary key.

- We insert student records into the `Students` table without specifying a value for `StudentID`, allowing the database to automatically generate unique identifiers for each record.

- The `StudentID` column serves as a simple, unique identifier for each student record, facilitating efficient querying, indexing, and referencing of student data.